- MISE AU PØINT -

Il y a des moments dans la vie où les choses ne vont pas forcément comme on veut. Çà a dû sûrement vous arriver si vous avez commencé à taper et à METTRE AU PØINT les exercices des pages précédentes.
METTRE AU PØINT un programme, çà a un sens très précis, çà veut dire

I

qu'un programme, avant d'être passé à l'épreuve d'un vrai ordinateur, c'est toujours faux, parce qu'on a oublié des trucs, ou alors parce qu'il y en a en trop, ou parce qu'on a tapé sur la carte à côté, ou bien qu'on avait perdu une carte sans s'en rendre compte, ou parceque le programme boucle ou encore pour un tas d'autres raisons.

II

que c'est TØUJØURS le programme qui est faux ; il ne faut jamais soupçonner la machine elle même, sans çà on ne pourrait plus jamais croire en rien. De toutes façons, il ne faut jamais prendre les ordinateurs pour des imbéciles. A priori, l'imbécile c'est soi-même.

III

qu'il n'y a pas de raison de rester imbécile TØUJØURS (ou plus précisément, de rester toujours un imbécile de même type); toutes les erreurs de programmation que j'ai dis en I, on peut les FAIRE SAUTER. Çà veut dire qu'on examine le programme, on repère où çà coince, on rectifie, on vérifie encore un coup, et on repasse le programme en machine.
Plus brièvement, çà signifie qu'on peut AMELIØRER
. la qualité d'un programme.
. sa propre technique.
La programmation a donc cette vertu de rompre le bon vieux principe en vigueur dans l'enseignement des trucs scientifiques, à savoir que le bonhomme qui a pas compris tout de suite, il comprendra jamais, et il n'a plus qu'à ramasser bonquier ou général.

L'Art de la programmation, c'est de repérer très vite où sont les erreurs qu'on a fait dans ses programmes (ce n'est pas tout l'Art, mais c'est une bonne [illegible] de l'Art).

- LA NOTION DE FONCTION -

Nous savons fort bien déjà de quoi il s'agit, et nous en connaissons beaucoup : CAR, CDR, CONS, PROG, NULL, COND, EQ, TIMES, etc.

Cependant, il n'existe pas en LISP de fonctions qui calcule la moyenne d'une liste de nombres (revoir la page 47), et qu'on appellerait ainsi, par exemple :

(MOYENNE '(3 21 -5 7 2))

et qui donnerait (3 + 21 - 5 + 7 + 2) / 5 → 28/5 → 5

ou encore ainsi :

soit L = (7 3 8)

(MOYENNE L) → 6

Or, si nous revoyons la page 47, nous constatons que NOUS SAVONS écrire un programme qui calcule la moyenne d'une liste de nombres. Le problème, c'est d'en faire une FONCTION qu'on appellerait exactement comme CAR, ou TIMES, ou QUO, à cette seule différence qu'on lieu d'être fournie par LISP, on l'aurait fabriquée soit-même.

Supposons que la liste de nombres soit dans une boîte LL, le morceau de programme qui en calculerait la moyenne pourrait être comme ceci :

morceau de programme

```
(PROG (N RES)                                          1
      (SETQ RES (SETQ N 0))
      (WHILE LL                                        2
          (SETQ N (ADD1 N))
          (SETQ RES (PLUS RES (NEXTL LL)))             3 2
      (RETURN (QUO RES N))                             1
```

Notre premier problème, c'est d'associer le nom choisi par nous, ici, "MOYENNE" à ce morceau de programme.
DÉFINIR une fonction, c'est, parmi bien d'autres choses, associer un NOM avec un morceau de programme. L'idée c'est que le morceau-de-programme, on ne l'aura écrit qu'une fois, mais, toutes les fois que son NOM sera en 1ère position d'un appel de fonction ex : (MOYENNE '(7 3 8)), ce serait comme si le morceau-de-programme surgissait et venait faire le travail.

Si notre idée réussissait, MØYENNE serait devenue une fonction comme une autre, fonction ici à 1 argument. Mais si on examine attentivement le morceau-de-programme de la page 59, on s'aperçoit que LL n'est pas déclarée dans la liste de variables du PRØG. En effet, LL est précisément l'argument de la nouvelle fonction MØYENNE. Nous allons voir tout çà s'éclaircir bientôt. Je prend de la hauteur.

DÉFINIR une fonction, c'est écrire :

(DE nom de la fonction liste d'arguments corps de fonction)

"DE" est une fonction qui me permet de définir d'autres fonctions. Elle a 3 arguments

1.) un nom choisi par vous-même (différent de celui des fonctions, du genre CAR etc. que LISP vous offre)

2.) une liste de boîte (de variables), avec l'usage qu'on va voir

3.) le "morceau-de-programme" de tout à l'heure que je baptise pompeusement "corps de fonction". Ce n'est qu'un baptême, on s'y fait très bien.

Armé de "DE" je définis alors la fonction "MØYENNE":

```
(DE MØYENNE (LL)
1
   (PRØG (N RES)
2
      (SETQ RES (SETQ N 0))
      (WHILE LL
3
            (SETQ N (ADD1 N))
            (SETQ RES (PLUS RES (NEXTL LL))) )
                                              3
      (RETURN (QUØ RES N)) ))
                            21
```

Après avoir écris ceci, j'ai le droit d'écrire, par exemple

(SETQ PAF (MØYENNE ⁹(7 3 8)))

et d'en attendre PAF

$\boxed{6}$

L'argument de MOYENNE, c'est LL. Avant de m'expliquer complètement, voici encore quelques exemples (on fond ce qui est le plus difficile dans la vie, c'est d'arriver à s'expliquer correctement).

• Je vais définir une fonction qui ajoute 10 à un argument, qui doit être un nombre :

```
(DE   PLUS10   (X)   (PLUS X 10))
      nom      liste  corps de
               d'arguments  fonction
```

Je donne ça à avaler à LISP qui dit, du coup qu'il est très content. Puis j'ènis :

```
(PLUS10  3)
```

Voici l'effet : ça place 3 dans la boîte nommée X

et ça évalue le corps de fonction. Tout se passe alors comme si on avait écrit (PLUS 3 10), et, comme vous l'avez deviné, ça ramène 13 comme VALEUR.

Puis j'ènis :

```
(PLUS10  -40)
```

Bien, dit LISP, il place

, et ramène, après évaluation de (PLUS X 10) → -30.

• Je définis à présent une fonction à 2 arguments (des nombres toujours), qui devrait me ramener la différence de leurs carrés.

```
(DE  DIFCAR  (M  N)
     (DIFFER  (TIMES M M)  (TIMES N N)))
```

Puis j'ènis :

```
(DIFCAR  10  3)
```

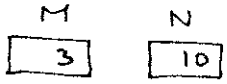
. Ça place (attention à l'ordre)

, et ça évalue le corps de fonction.

Ça ramène 10*10 - 3*3 → 91.

Si j'ènis à présent :

```
(DIFCAR  3  10)
```

, ça place 

et ça ramène : 3*3 - 10*10 → -91.

• Je définis une fonction à 1 argument, qui devra être une liste, et qui m'en ramène le nombre d'éléments.

```
(DE  LENGTH  (L)
     (PROG  (N)
            (SETQ  N  0)
            (WHILE  L  (SETQ  N  (ADD1  N))
                       (SETQ  L  (CDR  L)))
            (RETURN  N)))
```

Revoir pages 40 et 47.

Je puis écrire à présent :

(LENGTH '(DO SOL MI)) , ça place L (DO SOL MI) et ça passe la main au corps de fonction, qui fait le travail et me ramène 3 .

A présent réfléchissons un peu.
Au fond, il y a 2 sortes de façons d'être un ARGUMENT de fonction.

1er type : (DE PLUS1 (X) ...)
argument

2d type : (PLUS1 3)
argument

le 1er type apparaît dans les définitions de fonctions. L'argument est alors un nom de boîte (ou de variable)
Le 2d type apparaît dans les appels de fonctions. C'est alors une VALEUR qui sera automatiquement placée dans la boîte du 1er type.

C'est purement terminologique, on dit que les arguments de 1er type, (ceux de la liste-d'arguments du DE), ce sont des arguments FORMELS, et que ceux du 2d type, ceux qu'on place à la suite du nom, au moment de l'appel de la fonction, ce sont des arguments ACTUELS.

exemples :

- (DE PLUS1 (X) ...) arguments formels : X
- (DE DIFCAR (M N) ...) arguments formels : M N
- (DE LENGTH (L) ...) arguments formels : L

Resumons-nous : 1.) je définis une fonction au moyen d'un DE et tout l'attirail.
2.) je veux appeler la nouvelle fonction, que j'ai définie en 1.). Bon. Je l'appelle avec des arguments. Il y en aura peut-être plus d'un (comme dans la fonction DIFCAR). Les arguments actuels sont associés un par un avec les arguments formels, dans le même ordre d'apparition.

exemple : (DE FΦΦ (A B C) ...)

puis : (FΦΦ x_1 x_2 x_3) .

AVANT de passer la main au corps-de-fonction, LISP crée automatiquement les boîtes A B C ,

et place dedans : A [x_1] B [x_2] C [x_3] .

exemple : (DE FUU (L M) ...)

puis : (FUU 24 56)

AVANT d'évaluer le corps de fonction, LISP place

L [24] M [56] .

Si j'avais écris à la place : (FUU 56 24), même chose, LISP commence par :

L [56] M [24] .

CΦNCLUSIΦN PARTIELLE : et, quand l'exécution de la fonction est terminée, les boîtes-arguments-formels créés par l'appel sont automatiquement détruites. Elles perdent leur existence, peu à plus, finies les boîtes. C'est comme si elles n'avaient jamais existé.
On dit que l'appel de la fonction ALLΦUE (allocation) les boîtes-arguments-formels ; l'achèvement de l'exécution de l'appel de fonction les DÉSALLΦUE (désallocation).

EXEMPLES

- Une fonction qui inverse une liste

```
(DE REVERSE (L)
    (PRΦG (M)
        (SETQ M NIL)
        (WHILE L
            (SETQ M (CΦNS (NEXTL L) M)))
        (RETURN M) ))
```

```
(REVERSE '(A B C)) → (C B A)
(REVERSE '(JE (VΦUS AIME)) → ((VΦUS AIME) JE)
```

- Une fonction qui calcule l'élément maximum d'une liste de nombres

```
(DE MAX (L)
  (PROG (MX)
        (SETQ MX (NEXTL L))
        (WHILE L
          (COND
            ((GT (CAR L) MX)
                 (SETQ MX (CAR L)))
            (NEXTL L))
        (RETURN MX)))
```

```
(MAX '(1 2 3)) → 3
(MAX '(-4 0 1 2 -1 7 4 2)) → 7
```

- Une fonction qui ramène T si l'atome AT est élément d'une liste L, et qui ramène NIL sinon.

```
(DE MEMQ (AT L)
    (PROG ()
          (WHILE L
            (COND
              ((EQ AT (NEXL L))
                   (RETURN T)))
          (RETURN NIL)))
```

```
(MEMQ 3 '(6 7 3 5)) → T
(MEMQ 'JE '(IL VOUS AIME)) → NIL
```

- Une fonction qui ramène une copie de la liste L, d'où toutes les occurences de l'atome OBJ sont éliminées.

```
(DE DELQ (OBJ L)
    (PROG (RES)
          (WHILE L
            (COND
              ((NEQ (CAR L) OBJ)
                   (SETQ RES (CONS (CAR L) RES))))
            (SETQ L (CDR L)))
          (RETURN (REVERSE RES))))
```

Des fonctions définies peuvent s'appeller entre elles. DELQ appelle REVERSE, définie page 63.

```
(DELQ   5   '(1 5 3 4 5 2)) → (1 3 4 2)
(DELQ   'NE  '(JE NE VOUS AIME PAS)) → (JE VOUS AIME PAS)
(DELQ   'PAS  (DELQ  'NE  '(JE NE VOUS AIME PAS))) →
                                   (JE VOUS AIME)
```

- Une fonction qui compte le nombre d'occurences de l'atome AT dans une liste L.

```
(DE  COMBIEN  (AT  L)
     (PROG  (N)
            (SETQ  N  0)
            (WHILE  L
               (COND
                  ((EQ  (NEXTL  L)  AT)
                        (SETQ  N  (ADD1  N)))))
            (RETURN  N)))
```

```
(COMBIEN  'RE  '(SOL RE FA DO RE SOL DO FA RE)) → 3
(COMBIEN  'SOL  '(SOL RE FA DO RE SOL DO FA RE)) → 2
(COMBIEN  'LA  '(SOL RE FA DO RE SOL DO FA RE)) → 0
```

- Une fonction qui ramène la liste : (FINI POUR LE MOMENT)

```
(DE  OUF  ()  '(FINI POUR LE MOMENT))

(OUF) → (FINI POUR LE MOMENT)
```

OUF est ici une fonction à 0 arguments ; la liste d'arguments formels est donc ( ), c'est à dire NIL.

Relisez calmement tout ce qui précède à partir de la page 59, et étudiez très attentivement les fonctions REVERSE (page 63), MAX, MEMQ, DELQ (page 64), et COMBIEN (cette page). Au besoin, passez les en machine.

EXERCICES :

1 a.) écrire une fonction ABS à 1 argument (cet argument doit être un nombre) qui donne la valeur absolue de ce nombre.

ex : (ABS −15) → 15 , (ABS 1093) → 1093

b.) écrire une fonction SIGNE à 1 argument (numérique) qui livre : 1 si le nombre est > 0
0 si le nombre est = 0
−1 si le nombre est < 0

c.) écrire une fonction PAIR à 2 arguments. Ces 2 arguments devront être des listes de même longueur. PAIR devra ramener une liste de couples d'éléments de rang correspondant dans les 2 listes.

ex : X = (x_1 x_2 ... x_n)
Y = (y_1 y_2 ... y_n)
(PAIR X Y) → ((x_n y_n) ... (x_2 y_2)(x_1 y_1))

ex : X = (CHAT LE DØRT CIEL)
Y = (CAT THE SLEEPS SKY)
(PAIR X Y) →
((CIEL SKY)(DØRT SLEEPS)(LE THE)
(CHAT CAT))

Donc PAIR permet de construire des tables d'associations ou des dictionnaires simples.

d.) écrire une fonction (ASSØC objet liste de couples).
"objet" doit être un atome, "liste de couple" est une liste comme celles construites par PAIR.
Si "objet" est 1er element d'un des couples, ASSØC ramène le couple, sinon ASSØC ramène NIL

e.) écrire une fonction (RAVI prenom) qui ramène (HELLØ prenom RAVI DE VØUS VØIR)

http://www.artinfo-musinfo.org Initiation à la programmation en LISP 510, 1973-1977, page 71 / 130

SOLUTIONS :

1 a.)
```
(DE ABS (X)
    (COND
        ((LT X 0) (DIFFER 0 X))
        (T X)))
```

b.)
```
(DE SIGNE (X)
    (COND
        ((LT X 0) -1)
        ((ZEROP X) 0)
        (T 1)))
```

c.)
```
(DE PAIR (X Y)
    (PROG (RES)
        (SETQ RES NIL)
        (WHILE X
            (SETQ RES
                (CONS (LIST (NEXTL X) (NEXTL Y))
                      RES)))
        (RETURN RES)))
```

d.) voici à cette occasion une nouvelle fonction fort commode :
(CAAR l), qui fait le même travail que
(CAR (CAR l))

ex : (CAAR '((DIEU GOD) (DIABLE DEVIL))) → DIEU
(CAAR '((A) B)) → A
(CAAR '(((A) B) C) → (A)

```
(DE ASSOC (OB L)
    (PROG ()
        (WHILE L
            (COND
                ((EQ (CAAR L) OB)
                    (RETURN (CAR L))))
            (NEXTL L))
        (RETURN NIL)))
```

ex : (ASSOC 'CHAT '((NOIR BLACK)(CHAT CAT)(LE THE))) →
(CHAT CAT)
(ASSOC 4 '((1 UN)(2 DEUX))) → NIL

ASSOC permet donc de consulter un dictionnaire.

e.)
```
(DE RAVI (PRENOM)
    (CONS 'HELLO
        (CONS PRENOM
            '(RAVI DE VOUS VOIR))))
```

ex. : (RAVI 'HARALD) →
(HELLO HARALD RAVI DE VOUS VOIR)

- REPRESENTATIONS -

Que peut on faire avec des listes ?

. représenter des objets
. exprimer diverses relations entre ces objets
. manipuler ces relations

Représentations, modèles, copies, simulacres, caverne et République V et VI, immenses problèmes. Ce n'est pas ici le lieu de les border. Je me contenterai d'énumérer des exemples.

On peut avec des listes représenter des ENSEMBLES, et des opérations sur des ensembles :

ex : A = (1 3 7 6)
B = (4 6 5 3 1)

l'union de A et B = (1 3 4 5 6 7)
l'intersection de A et B : (1 3 6)
le produit cartésien
de A et B : ((1 4)(1 6)(1 5)(1 3)(1 1)
(3 4)(3 6)(3 5)(3 3)(3 1)
(7 4)(7 6)(7 5)(7 3)(7 1)
(6 4)(6 6)(6 5)(6 3)(6 1))

Je peux également représenter commodément des PARTITIONS d'ensembles

ex : liste des partitions d'un ensemble à 3 éléments :

(((1)(2)(3)) ((1 2)(3))
((1 3)(2)) ((1)(2 3)) ((1 2 3)))

Je rappelle qu'établir la partition d'un ensemble, c'est répartir les éléments de cet ensemble en classes telles que . chaque element est dans une classe
. les classes sont DISJOINTES, i.e. n'ont pas d'éléments en commun.

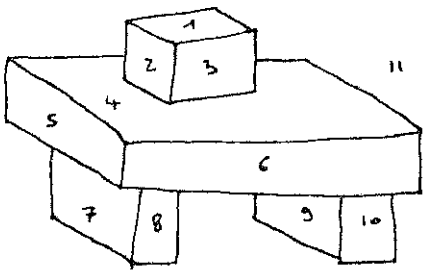
exemple : liste des partitions d'un ensemble à 4 éléments

```
(  ((1)(2)(3)(4))
   ((1 2)(3)(4))
   ((1 3)(2)(4))
   ((1 4)(2)(3))
   ((1)(2 3)(4))
   ((1 2 3)(4))
   ((1 4)(2 3))
   ((1)(2 4)(3))
   ((1 2 4)(3))
   ((1 3)(2 4))
   ((1)(2)(3 4))
   ((1 2)(3 4))
   ((1 3 4)(2))
   ((1)(2 3 4))
   ((1 2 3 4))  )
```

Application des partitions d'ensembles à la description de scènes visuelles :

Voici 4 blocs ainsi posés les uns sur les autres (représentation bi-dimensionnelle d'une scène 3-dimensionnelle) :

j'ai numéroté les faces.



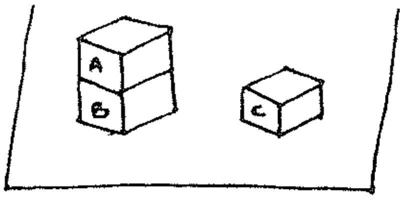
Reconnaître la structuration de cette scène en 4 blocs, c'est établir la partition que voici d'1 ensemble à 11 éléments :

```
(  (1 2 3)
   (4 5 6)
   (7 8)
   (9 10)
   (11)  )
```

Il existe d'autres manières de représenter de telles scènes, par exemple au moyen d'une liste de FAITS ou ITEMS ou PREDICATS.

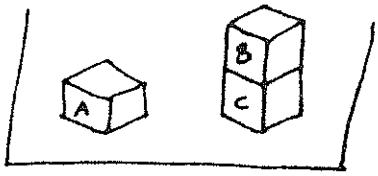
ex :

```
( (SUR A B)
  (SUR B TABLE)
  (SUR C TABLE))
```



On peut se demander quelles sont les manipulations qui permettraient d'obtenir, à partir de la précédente, la scène transformée :

```
( (SUR A TABLE)
  (SUR B C)
  (SUR C TABLE))
```



en respectant les règles que voici :

. on ne peut déplacer qu'une boîte "libre" (rien posé dessus)

. ne placer une boîte sur une autre que si cette dernière est libre.

On peut également décrire des scènes à 2 dimensions, par exemple en utilisant les prédicats :

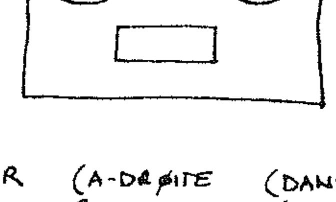
```
(SØUS x y)      i.e.  x est sous y
(DANS x y)      i.e.  x contenu dans y
(A-DRØITE x y)  i.e.  x est à la droite de y.
```

et les symboles d'objets :

```
T : triangle
R : rectangle
C : cercle
```

On codera ainsi des images simples

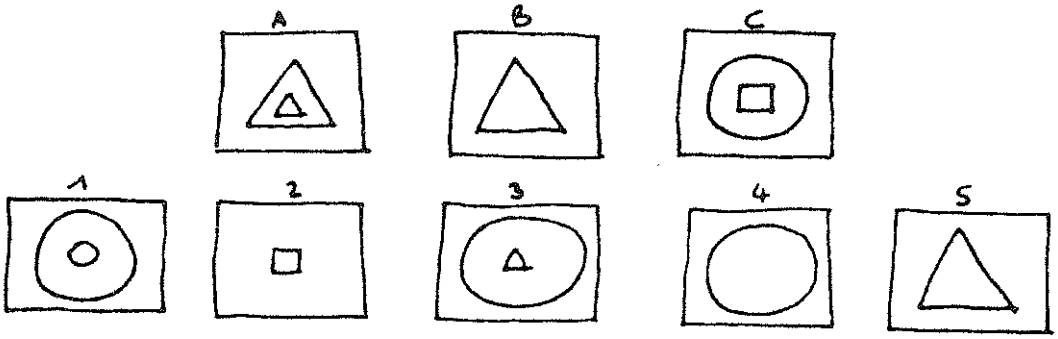
ex :



```
(DANS (SØUS R (A-DRØITE (DANS T C)(DANS R C))) R)
 1     2       3         4      4 4      4 3 2  1
```

En fait, il s'agit moins du codage de l'image proprement dite, que de la représentation symbolique de sa STRUCTURE RELATIØNNELLE.

Autre exemple : voici 8 images : A, B, C, 1, 2, 3, 4, 5.
Je dis que A correspond avec B.
Quelle est, parmi 1, 2, 3, 4, 5 celle qui correspond avec C.
(c'est le test d'ANALOGIE bien connu)



Solution : 4 bien sûr. Il existe (EVANS) un programme capable de donner la solution correcte à un grand nombre de problèmes de cette famille.

EXERCICE : représenter symboliquement les images A, B, C, 1, 2, 3, 4, 5, puis tirer de ces représentations symboliques, de bonnes raisons, sachant que A est transformé en B, de dire que c'est en 4 que C doit être transformé pour respecter l'analogie.

SOLUTION :

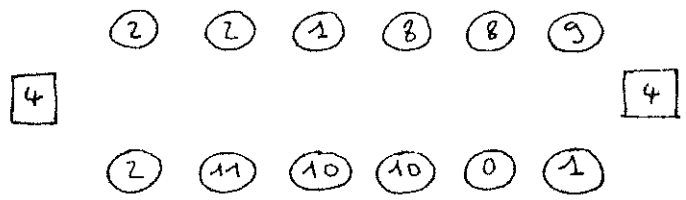
A : (DANS (DANS T T) R)
B : (DANS T R)
C : (DANS (DANS R C) R)
~~B~~ : (DANS (DANS C C) R)
2 : (DANS R R)
3 : (DANS (DANS T C) R)
4 : (DANS C R)
5 : (DANS T R)

A : (DANS (DANS T T) R)
↓
B : (DANS T R)

C : (DANS (DANS R C) R)
↓
4 : (DANS C R)

Les listes se prêtent également assez bien aux représentations de JEUX.

ex : Voici la situation des 2 joueurs à un moment précis d'une partie de KALAH (ou AWELE)




```
( (2  11  10  10  0  1  (4))
  (9  8   8   1   2  2  (4)) )
```

Restons dans les jeux. Voici, sous la forme d'une liste, une représentation symbolique d'une (bonne) main de PØKER.

((AS PIQUE)(AS CŒUR)(AS CARREAU)(AS TREFLE)(7 CŒUR))

EXERCICE : Ecrire un programme qui imprime toutes les cartes d'un jeu de 52 cartes, en utilisant les 2 listes :

FIGURE : (AS RØI DAME VALET 10 9 8 7)
CØULEUR : (PIQUE CŒUR CARREAU TREFLE)

i.e. :
(AS PIQUE)
(AS CŒUR)
(AS CARREAU)
(AS TREFLE)
(RØI PIQUE)
(RØI CŒUR)
etc ...

SØLUTIØN :

```
(DE CARTES ()
  (PRØG (FIGURE FIG CØULEUR)
        (SETQ FIGURE '(AS RØI DAME VALET 10 9 8 7))
        (WHILE FIGURE
          (SETQ FIG (NEXTL FIGURE))
          (SETQ CØULEUR '(PIQUE CŒUR CARREAU TREFLE))
          (WHILE CØULEUR
            (PRINT (LIST FIG (NEXTL CØULEUR))) ) )
        (RETURN '(YEN A PLUS)) ))
```

SUGGESTION : Un programme qui jouerait au POKER (ça existe, celui de WATERMAN) devrait (c'est la moindre des choses) reconnaître la valeur de sa main

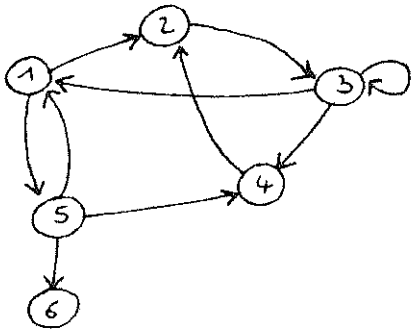
ex : reconnaître 1 brelan, 1 quinte, une suite, 2 pairs etc.

Essayez de concevoir une fonction qui aurait comme argument une liste de 5 cartes, et qui renverrait sa valeur

ex : brelan d'as, carré de dames, paire de 10, etc ...

On peut, avec des listes représenter des GRAPHES

ex. :



((2 5) (3) (1 3 4) (2) (1 4 6) ())

Ici, le n ième element de la liste représente la liste des sommets (les "points" du graphe) qui peuvent être atteints en 1 étape à partir de n.

EXERCICE : écrire une fonction (NIÈME N L) qui renvoie le N ième élement de la liste L.

ex : (NIEME 3 '(A B C D)) → C
(NIEME 2 '(1 (3 4) 2 6)) → (3 4)

SOLUTION :

```
(DE NIEME (N L)
   (PROG ()
      (WHILE (GET N 1)
             (NEXTL L)
             (SETQ N (SUB1 N)) )
      (RETURN (CAR L)) ))
```

Les listes permettent également de représenter symboliquement des objets musicaux.

ex : étant donné l'ensemble d'objets

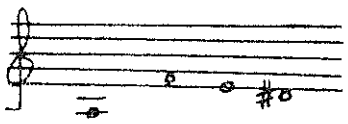
(DØ RE MI FA SOL LA SI)

on regroupe (si c'est possible) ces objets en accords parfaits

((FA LA DØ) (MI SOL SI) RE)

On représente commodément des ALTERNATIVES

ex :



(LA FA MI (# RE))

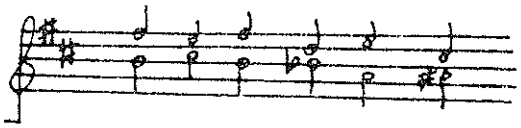
On pourra associer à chaque nom de note, sa représentation numérique :

((LA 9) (FA 5) (MI 4) ((# RE) 3))

par le codage :

SI #	RE b	RE	MI b		DØb
DØ	DØ#		RE#		SI
↕	↕	↕	↕	...	↕
0	1	2	3		11

exemple : le début d'une description harmonique de



Ce sera, à partir de la liste initiale (donnée)

(SI (# FA) (# DØ) MI (% SI) RE LA (# LA))

obtenir, avec les relations suivantes :

D : dominante
SD : sous-dominante
T : triton
S : sensible
N : napolitaine

((D (SI (# FA)) ((# FA) (# DØ)) (MI SI) (RE LA))
(SD ((# FA) SI) ((# DØ) (# FA)) (SI MI) (LA RE))
(T ((% SI) MI) (MI (% SI)) ((# LA) MI) (MI (# LA)))
(S (RE (# DØ)) (SI (# LA)))
(N ((# DØ) RE) ((# LA) SI)))

Les listes peuvent également représenter des SUITES ou CHAÎNES.

ex : A = (JE SUIS RAVI)

B = (DE VOUS VOIR)

Une opération intrinsèquement définie sur des chaînes s'appelle la CONCATÉNATION (mettre bout à bout).

ex :

A concaténé avec B : (JE SUIS RAVI DE VOUS VOIR)

B concaténé avec A : (DE VOUS VOIR JE SUIS RAVI)

Intuitivement, avec les listes, l'idée c'est :

$(x_1\ x_2\ \dots\ x_n)\ (y_1\ y_2\ \dots\ y_m)$

$\downarrow$

$(x_1\ x_2\ \dots\ x_n\ y_1\ y_2\ \dots\ y_m)$

Voici 2 nouvelles fonctions LISP permettant d'utiliser cette opération.

(NCONC l_1 l_2) : concatène les 2 listes l_1 et l_2

(NCONC1 l ob) : place l'objet ob à la fin de la liste l.

exemples :

(NCONC '(A B) '(C)) → (A B C)
(NCONC1 '(A B) 'C) → (A B C)
(NCONC1 '(A B) '(C)) → (A B (C))

(NCONC '(JE (NE)) '(VOUS AIME (PAS))) → (JE (NE) VOUS AIME (PAS))
(NCONC '(JE VOUS AIME) '(BEAUCOUP)) → (JE VOUS AIME BEAUCOUP)
(NCONC1 '(JE VOUS AIME) '(BEAUCOUP)) → (JE VOUS AIME (BEAUCOUP))

remarque : (NCONC NIL '(A B)) → (A B)
(NCONC '(A B) NIL) → (A B)

NIL (ou ()) est bien "l'élément neutre" pour la concaténation.

EXERCICE : Ecrire une fonction (LAST L) qui renvoie le dernier élément de la liste L.

SOLUTION :

```
(DE LAST (L)
  (PROG (RES)
    (SETQ RES NIL)          : pour le cas où L serait
    (WHILE L                  vide.
      (SETQ RES (NEXTL L)))
    (RETURN RES) ))
```

EXERCICE : Faire un programme qui

- lit une liste-dictionnaire (voir pages 65 et 66), à l'allure ((LE THE)(CHAT CAT) ...)
- puis lit des phrases en français et en donne une "traduction" mot à mot en anglais.
(Quand le mot français ne figure pas dans le dictionnaire, laissez le mot français).

Utilisez les fonctions NCONC1 et ASSOC (que vous définirez vous mêmes, voir page 67).

SOLUTION :

```
(PROG (DICT PHR1 PHR2 MOT AUX)
      (SETQ DICT (READ))
      (WHILE (SETQ PHR1 (READ))
             (WHILE PHR1
                    (SETQ MOT (NEXTL PHR1))
                    (SETQ AUX (ASSOC MOT DICT))
                    (SETQ PHR2
                          (NCONC1 PHR2
                                  (COND
                                   (AUX (CADR AUX))
                                   (T MOT))))))
             (PRINT PHR2))
      (RETURN '(FIN DE TRADUCTION)))
```

Exemple : j'indique par "→" ce qui est lu par l'ordinateur et par "←" ce que l'animal imprime.

```
→ ((FILLE GIRL)(POMME APPLE)(MANGE EATS)(LA THE)(JEUNE YOUNG)(LE THE)
   (GRANDE BIG)(TUE KILLS))
→ (LA POMME MANGE LA FILLE)
← (THE APPLE EATS THE GIRL)
→ (LA GRANDE POMME TUE LA JEUNE POMME)
← (THE BIG APPLE KILLS THE YOUNG APPLE)
→ (LA JEUNE ZIZOU MANGE LE LOUP)
← (THE YOUNG ZIZOU EATS THE LOUP)
→ NIL
← (FIN DE TRADUCTION)
```

LISP permet également de représenter assez facilement des objets linguistiques

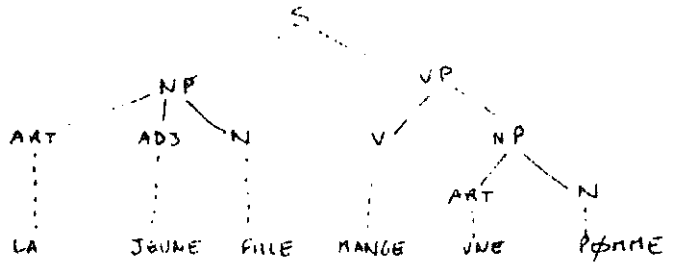
exemple : voici une phrase (on la trouve partout)

(LA JEUNE FILLE MANGE UNE PØMME)

Comment obtenir la liste suivante ?

```
( S   (NP (ART LA)(ADJ JEUNE)(N FILLE))
      (VP (V MANGE)
          (NP (ART UNE)(N PØMME)) ) )
```

qui représente symboliquement l'analyse



On peut concevoir plusieurs stades de construction

STADE 0 : (LA JEUNE FILLE MANGE UNE PØMME)

STADE 1 : ((ART LA)(ADJ JEUNE)(N FILLE)(V MANGE)
(ART UNE)(N PØMME))

STADE 2 : ((NP (ART LA)(ADJ JEUNE)(N FILLE))
(V MANGE)
(NP (ART UNE)(N PØMME)))

STADE 3 : ((NP (ART LA)(ADJ JEUNE)(N FILLE))
(VP (V MANGE)
(NP (ART UNE)(N PØMME))))

STADE 4 : (S (NP ...

Il existe un programme, celui de Terry WINØGRAD, qui fait un travail assez semblable pour des phrases d'un anglais (pas trop) restreint. De plus, si ces phrases se rapportent à un univers de blocs posés les uns sur les autres (voir pages 69 et 70), le programme comprend ce qu'on lui dit en anglais, et agit en conséquence.

« Programme, place donc la boîte qui soutient A, sur la boîte C »

Comment représenter symboliquement des règles d'une grammaire ?
Voici, très simplifié, la représentation de SIMMONS.

Définissons le langage suivant :

$$\{ SN \ (que \ SN)^n \ V^n \ V \mid n \gg 0 \}$$

Appartiendront à ce langage des phrases telles que :

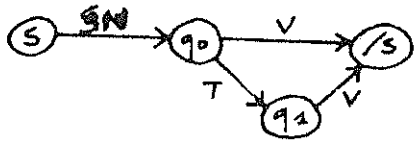
- LE FROMAGE PUE
- LE FROMAGE PUE
  QUE LE RAT RONGE
- LE FROMAGE PUE
  QUE LE RAT RONGE
  QUE LE CHAT CHASSE

etc.

Voici, de ce langage, une grammaire

S → SN + V
S → SN + T + V
T → que + S
SN → art + N
art → la, ...
N → fromage, rat, chat, ...
V → pue, ronge, chasse, ...

Cette grammaire est représentable par le graphe suivant (WOODS)



Voici alors, sous forme de listes la représentation symbolique de la grammaire.

```
( S   (SN  q0))
(q0   (T   q1) (V   /S))
(q1   (V   /S))
(T    (WH- q2))
(q2   (S   /T))
(SN   (ART q3))
(q3   (N   /SN))
```

et des classes :

```
(WH-   QUE)
(ART   LE  ...)
(N   FROMAGE  RAT  CHAT)
(V   PUE  RONGE  CHASSE ...)
```

Comme nous l'avons vu à propos de DICTIONNAIRES, les listes permettent de représenter commodément des tables.

exemple :

prénom	âge
JEAN	24
ALI	19
ALINE	23
NAIMA	27
LUC	22
CLAUDE	29
VICTOR	24
JEAN	31
CLAUDE	28

```
( (JEAN 24) (ALI 19) (ALINE 23) (NAIMA 27) (LUC 22) (CLAUDE 29)
  (VICTOR 24) (JEAN 31) (CLAUDE 28) )
```

On peut également, sachant que CLAUDE est un prénom tout aussi bien masculin et féminin, se préoccuper du sexe de ces personnes (H = homme, F = femme).

```
( (JEAN 24 H) (ALI 19 H) (ALINE 23 F) (NAIMA 27 F) (LUC 22 H)
  (CLAUDE 29 F) (VICTOR 24 H) (JEAN 31 H) (CLAUDE 28 H) )
```

A partir de telles tables, ou comme on dit quelquefois FILE ou FICHIER, on peut essayer de collecter des informations.

exemples :

- imprimer toutes les personnes de plus de 24 ans.

  ((NAIMA 27 F) (CLAUDE 29 F) (JEAN 31 H) (CLAUDE 28 H))

- imprimer toutes les femmes.

  ((ALINE 23 F) (NAIMA 27 F) (CLAUDE 29 F))

- imprimer toutes les personnes ayant même prénom

  ((JEAN 24 H) (CLAUDE 29 F) (JEAN 31 H) (CLAUDE 28 H))

- imprimer toutes les personnes du même age.

  ((JEAN 24 H) (VICTOR 24 H))

- imprimer toutes les femmes de moins de 25 ans

  ((ALINE 23 F))

- classer les personnes par âge croissant (TRI).

```
( (ALI 19 H) (LUC 22 H) (ALINE 23 F) (JEAN 24 H) (VICTOR 24 H)
  (NAIMA 27 F) (CLAUDE 28 H) (CLAUDE 29 F) (JEAN 31 H) )
```

etc...

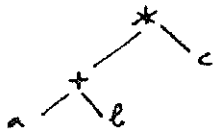
Une vieille question qu'on se pose, à partir de l'interrogation de files, c'est : peut-on à l'avance prévoir TOUTES les questions qu'on peut poser à un fichier ?

Une autre application bien connue des listes est la représentation d'expressions formelles logico-mathématiques.

exemple : soit l'expression

$$((a + b) * c)$$

on peut en construire "l'arbre"



La structure d'une expression de base serait :

(operande1 operateur operande2)

Mais, si je leur donne la structure :

(operateur operande1 operande2)

i.e. opérateur en tête : j'utilise la représentation dite POLONAISE PREFIXE : ici : (* (+ a b) c)

Si, d'autre part, j'impose la structure

(operande1 operande2 operateur)

i.e. operateur en queue : j'utilise la représentation dite POLONAISE POSTFIXE : ici : ((a b +) c *)

exemples : → soit : ((a + b) * (c - d))

PREFIXE : (* (+ a b) (- c d))

POSTFIXE : ((a b +) (c d -) *)

→ soit : (a + ((b - c) * d))

PREFIXE : (+ a (* (- b c) d))

POSTFIXE : (a ((b c -) d *) +)

REMARQUE : à l'apparence externe de l'opérateur près

i.e. + ou PLUS, - ou DIFFER, * ou TIMES etc.

il n'y a aucune différence entre

(* (+ a b) c) et (TIMES (PLUS A B) C)

Donc, LISP s'écrit NATURELLEMENT en représentation polonaise préfixe.

LISP permet donc la manipulation d'expressions algébriques sous forme symbolique.

exemple : passer de : $\int x e^x dx$ à $e^x (x-1) + C$

c'est transformer l'expression symbolique :

(INTGR (TIMES X (EXP E X)) D X)

en :

(PLUS (TIMES (EXP E X) (DIFFOR X 1)) CONST)

à savoir une expression d'où "INTGR" a disparu.

Plusieurs programmes écrits en LISP (SAINT de Slagle, SIN de Joel Moses) permettent l'intégration symbolique d'une large classe de fonctions.

CONCLUSION PARTIELLE : (voir page 58)

L'Art de la programmation, c'est aussi de choisir une bonne représentation en machine de son problème.

http://www.artinfo-musinfo.org Initiation à la programmation en LISP 510, 1973-1977, page 86 / 130

- FONCTIONS RECURSIVES -

Voici un petit problème : fabriquer une liste qui contienne n fois l'atome "DO".

Une façon de faire c'est, avec par exemple n = 3

(CONS 'DO (CONS 'DO (CONS 'DO NIL)))
1 2 3 321

J'évalue d'abord le 3ème cons, ça me donne :

(CONS 'DO (CONS 'DO '(DO)))

Puis j'évalue le 2d CONS, ce qui donne :

(CONS 'DO '(DO DO))

Enfin j'évalue le CONS le plus extérieur (c'est le seul qui reste) ; il donne :

(DO DO DO)

- Résumons-nous :

(CONS 'DO (CONS 'DO (CONS 'DO NIL)))
↓
(CONS 'DO (CONS 'DO '(DO)))
↓
(CONS 'DO '(DO DO))
↓
(DO DO DO)

avec n = 4 on aura :

(CONS 'DO (CONS 'DO (CONS 'DO (CONS 'DO NIL))))
↓
(CONS 'DO (CONS 'DO (CONS 'DO '(DO))))
↓
(CONS 'DO (CONS 'DO '(DO DO)))
↓
(CONS 'DO '(DO DO DO))
↓
(DO DO DO DO)

On tient l'idée. L'idée c'est d'accumuler les

(CONS 'DO ...

Jusqu'à

(CONS 'DO NIL)

puis d'évaluer effectivement les CONS ainsi emboîtés, en commençant par <u>le plus intérieur</u>.

Donc 2 étapes dans le travail

1. accumuler les (CONS 'DO ... jusqu'à (CONS 'DO NIL) où on arrête d'accumuler : DESCENTE ou PLONGÉE.
2. évaluer les CONS ainsi accumulés plus profond ou plus extérieur EMPONTÉE.

Ce travail : DESCENTE suivie de REMONTÉE est effectué par la fonction que voici :

```
(DE CONSD (N)
    (COND
        ((ZEROP N) NIL)
        (T ('DO (CONSD (SUB1 N))))))
```

Faisons tourner :

I. (CONSD 0) : ça place 0 dans N, et ça attaque le COND. Puisque N = 0, on ramène NIL. En effet, () est bien (et pour cause) une liste contenant 0 fois DO.

II. (CONSD 1) : ça place 1 dans N, et ça attaque le COND. Puisque N ≠ 0 ça appelle :

```
('DO (CONSD 0))
```

↑ valeur de (SUB1 N) avec N = 1.

Un ('DO ... est ainsi accumulé ; et ça appelle (CONSD 0). Reportons nous à I, (CONSD 0) → NIL. Donc on a :

(DO NIL) → (DO). Vu ?

III. (CONSD 2) : ça place 2 dans N, et ça attaque le COND. Puisque N ≠ 0, ça appelle :

```
('DO (CONSD 1))
```

↑ valeur de (SUB1 N) avec N = 2.

ça accumule ('DO ... et ça appelle (CONSD 1). Reportons nous à II, (CONSD 1) → (DO). Donc on a : ('DO '(DO)) → (DO DO). Voilà.

Résumons-nous :

↓ (CONSD 0)
NIL

OK

↓ (CONSD 1)
↓ ('DO (CONSD 0))
↓ ('DO NIL)
↓ (DO)

OK

```
(CONSD 2)
↓ (CONS 'DO (CONSD 1))                          DESCENTE
↓ (CONS 'DO (CONS 'DO (CONSD 0)))
↓ (CONS 'DO (CONS 'DO NIL))          ARRÊT DE LA DESCENTE
↓ (CONS 'DO '(DO))                   REMONTÉE
↓ (DO DO)
```

```
(CONSD 3)
↓ (CONS 'DO (CONSD 2))
↓ (CONS 'DO (CONS 'DO (CONSD 1)))
↓ (CONS 'DO (CONS 'DO (CONS 'DO (CONSD 0))))
↓ (CONS 'DO (CONS 'DO (CONS 'DO NIL)))
↓ (CONS 'DO (CONS 'DO '(DO)))
↓ (CONS 'DO '(DO DO))
↓ (DO DO DO)
```

Notez bien que si j'avais défini CONSD ainsi :

```
(DE CONSD (N)
    (CONS 'DO (CONSD (SUB1 N))))
```

(à n'avoir pas essayé :

```
(CONSD 1)
↓ (CONS 'DO (CONSD 0))
↓ (CONS 'DO (CONS 'DO (CONSD -1)))
↓ (CONS 'DO (CONS 'DO (CONS 'DO (CONSD -2))))
```

etc. ad infinitum.

La plongée ne se serait jamais terminée. Car N se diminuait bien de 1 à chaque appel, mais le test vérifiant que N = 0 manquant, on n'avait aucune possibilité de stopper la plongée (l'accumulation en mémoire NIL, ce qui permettait la remontée.

DONC, dans la bonne version de CONSD (page 83), on arrête la plongée quand N = 0, ce qui est testé par le (ZEROP N), et à partir de ce moment, on "remonte" en CONSsant pour de bon et à mesure les DO.

Relisez tranquillement les pages 82, 83 et 84.

On dit qu'une fonction est RÉCURSIVE, lorsque, dans son corps-de-fonction, la fonction s'appelle elle même.
C'est fort simple, il faut 1/ plonger.
2./ arrêter la plongée si un certain test est vérifié.
3./ remonter.
C'est encore une forme d'ITÉRATION (voir page 52).

Voici un autre exemple de fonction récursive : une fonction qui compte le nombre d'éléments d'une liste :

```
(DE LENGTH (L)
    (COND
        ((NULL L) 0)
        (T (ADD1 (LENGTH (CDR L))))))
```

Essayons :

OK :

```
(LENGTH NIL)
↓
0                    OK.
```

```
(LENGTH '(A))
↓
(ADD1 (LENGTH NIL))
↓
(ADD1 0)
↓
1                    OK.
```

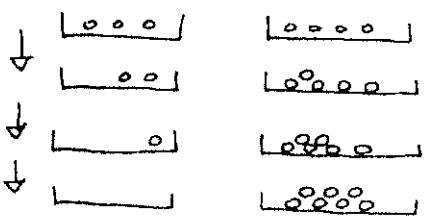
```
(LENGTH '(A B))
↓
(ADD1 (LENGTH '(B)))
↓
(ADD1 (ADD1 (LENGTH NIL)))
↓
(ADD1 (ADD1 0))
↓
(ADD1 1)
↓
2
```

```
(LENGTH '(A B C))
↓
(ADD1 (LENGTH '(B C)))
↓
(ADD1 (ADD1 (LENGTH '(C))))
↓
(ADD1 (ADD1 (ADD1 (LENGTH NIL))))
↓
(ADD1 (ADD1 (ADD1 0)))
↓
(ADD1 (ADD1 1))
↓
(ADD1 2)
↓
3
```

Vu ?

Encore un exemple. Voici une façon très spéciale de faire l'addition de 2 nombres. Il faut seulement être capable d'ajouter 1, d'enlever 1, et de tester si un nombre est égal à zéro. Je vous en livre le principe par les dessins suivants :

ex : 3 + 4.



i.e. j'enlève à chaque étape un caillou dans la boîte de gauche, que je place dans la boîte de droite. Quand ya plus de cailloux à gauche, j'ai la somme des cailloux à droite.
Il faut donc pouvoir compter à l'endroit : 0, 1, 2, 3, 4, 5, ...
et à l'envers : ... 6, 5, 4, 3, 2, 1, 0. (A l'école élémentaire on devrait AUSSI apprendre aux enfants à compter à l'ENVERS).

Voici ma fonction :

```
(DE RPLUS (X Y)
    (CØND
      ((ZERØP X) Y)
      (T (RPLUS (SUB1 X)(ADD1 Y))) ))
```

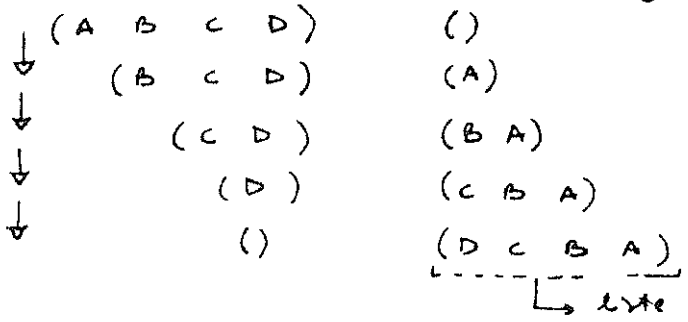
Tournons :

```
• (RPLUS 0 2)
  2                ØK.

• (RPLUS 1 2)
  (RPLUS 0 3)
  3                ØK.

• (RPLUS 3 4)
  (RPLUS 2 5)
  (RPLUS 1 6)
  (RPLUS 0 7)
  7                ØK.

• (RPLUS 5 2)
  (RPLUS 4 3)
  (RPLUS 3 4)
  (RPLUS 2 5)
  (RPLUS 1 6)
  (RPLUS 0 7)
  7                ØK.
```

Autre exemple : inverser une liste. Voici, graphiquement l'idée :

```
(A B C D)      ()
  (B C D)      (A)
    (C D)      (B A)
      (D)      (C B A)
        ()     (D C B A)   STØP.
```

↳ liste inversée.



i.e. le CAR de la liste de gauche, je le CØNSe à la liste de droite qui est toujours NIL pour commencer.

```
(DE REVERSE (L M)
  (CØND
    ((NULL L) M)
    (T (REVERSE (CDR L)
                (CØNS (CAR L) M) ))))
```

Faisons tourner :

```
⊙ (REVERSE NIL NIL)
  NIL          ØK.

⊙ (REVERSE '(A B) NIL)
  (REVERSE '(B) (CØNS 'A NIL))
  (REVERSE '(B) '(A))
  (REVERSE NIL (CØNS 'B '(A)))
  (REVERSE NIL '(B A))
  (B A)        ØK

⊙ (REVERSE '(A B C) NIL)
  (REVERSE '(B C) (CØNS 'A NIL))
  (REVERSE '(B C) '(A))
  (REVERSE '(C) (CØNS 'B '(A)))
  (REVERSE '(C) '(B A))
  (REVERSE NIL (CØNS 'C '(B A)))
  (REVERSE NIL '(C B A))
  (C B A)      ØK
```

Faire tourner ces fonctions comme je l'ai fait, c'est effectuer ce que nous informaticiens nommons une TRACE.

En page 80, nous avons parlé d'EXPRESSIONS POLONAISES, préfixes ou postfixe (Relisez).

EXERCICES : donner la forme polonaise préfixe de

a.) (A + B)

b.) ((A + B) * (A - B))

c.) (A * ((B - C) + (D / (E - F))))

SOLUTIONS :

a.) (+ A B)

b.) (* (+ A B) (- A B))

c.) (* A (+ (- B C) (/ D (- E F))))

Mais aussi bien A, ou C sont des expressions.

Donc une expression E c'est : . ou bien un atome
. ou bien une liste à 3 éléments

$$E = (\text{operande}_1 \quad \text{operateur} \quad \text{operande}_2)$$

i.e. (CAR E) : operande1
(CADR E) : operateur
(CADDR E) : operande2

et les opérandes sont eux-mêmes des expressions.

donc EXPRESSION c'est . atome
ou
(EXPRESSION operateur EXPRESSION).

Définition elle même récursive.

Voici une fonction qui a comme argument une expression, et qui en donne la forme polonaise préfixe.

```
(DE   POL  (E)
   (COND
      ((ATOM E) E)
      (T (LIST   (CADR E)
                 (POL (CAR E))
                 (POL (CADDR E)) )) ))
```

Tournons :

```
• (POL   '(A + B))
  (LIST  '+  (POL 'A)  (POL 'B))
  (LIST  '+  'A   'B)
  (+ A B)        OK
```

```
⊙ (PØL   '((A + B) * C))
↓ (LIST  '*  (PØL '(A + B))  (PØL 'C))
↓ (LIST  '*  (LIST '+ (PØL 'A)(PØL 'B))  (PØL 'C))
↓ (LIST  '*  (LIST '+ 'A 'B)  (PØL 'C))
↓ (LIST  '*  '(+ A B)  (PØL 'C))
↓ (LIST  '*  '(+ A B)  'C)
↓ (*  (+ A B)  C)        ØK
```

Voici à présent une fonction qui évalue des expressions arithmétiques préfixes.

```
(DE EVPØL (E)
   (CØND
     ((ATØM E) E)
     ((EQ (CAR E) '+)
        (PLUS (EVPØL (CADR E)) (EVPØL (CADDR E))))
     ((EQ (CAR E) '-)
        (DIFFER (EVPØL (CADR E)) (EVPØL (CADDR E))))
     ((EQ (CAR E) '*)
        (TIMES (EVPØL (CADR E)) (EVPØL (CADDR E))))
     ((EQ (CAR E) '/)
        (QUØ (EVPØL (CADR E)) (EVPØL (CADDR E))))))
```

La nouvelle fonction :

(ATØM obj) ⟶ [T si l'objet obj est 1 atome ; NIL sinon]

exemples :

```
(ATØM 4) → T
(ATØM NIL) → T
(ATØM 'ØK) → T
(ATØM '(A B)) → NIL
(ATØM '(NIL)) → NIL
```

faisons tourner EVPØL.

```
• (EVPØL '(+ (* 2 5) (- 8 3)))
↓ (PLUS (EVPØL '(* 2 5)) (EVPØL '(- 8 3)))
↓ (PLUS (TIMES (EVPØL 2) (EVPØL 5)) (EVPØL '(- 8 3)))
↓ (PLUS (TIMES 2 5) (EVPØL '(- 8 3)))
↓ (PLUS 10 (EVPØL '(- 8 3)))
↓ (PLUS 10 (DIFFER (EVPØL 8)(EVPØL 3)))
↓ (PLUS 10 (DIFFER 8 3))
↓ (PLUS 10 5)
↓ 15        ØK.
```

EXERCICES :

[1] : Relisez p. 67, et donnez une version récursive de la fonction ASSOC.

[2] : Ecrire une fonction récursive (DELQ at l) qui donne une copie de la liste l d'où toutes les occurences de l'atome at ont été éliminées.

[3] : Relisez la page 68. En considérant les listes comme des ENSEMBLES, voir la fonction :

(MEMQ at l) → { T si at est élément de la liste l ; NIL sinon }

Ecrire des fonctions récursives (UNION L1 L2) et (INTER L1 L2) qui livrent respectivement l'union et l'intersection des listes L1 et L2.

SOLUTIONS :

[1]

```
(DE ASSOC (OBJ L)
    (COND
        ((NULL L) NIL)
        ((EQ (CAAR L) OBJ)
            (CAR L))
        (T (ASSOC OBJ (CDR L)))))
```

[2]

```
(DE DELQ (AT L)
    (COND
        ((NULL L) NIL)
        ((EQ (CAR L) AT)
            (DELQ AT (CDR L)))
        (T (CONS (CAR L)
                 (DELQ AT (CDR L))))))
```

. Faisons tourner DELQ.

```
⊙ (DELQ 'B '(A B C B))
↓ (CONS 'A (DELQ 'B '(B C B)))
↓ (CONS 'A (DELQ 'B '(C B)))
↓ (CONS 'A (CONS 'C (DELQ 'B '(B))))
↓ (CONS 'A (CONS 'C (DELQ 'B NIL)))
↓ (CONS 'A (CONS 'C NIL))
↓ (CONS 'A '(C))
↓ (A C)        OK. Tous les "B" ont été éliminés.
```

3

```
(DE UNIØN (L1 L2)
    (CØND
        ((NULL L1) L2)
        ((MEMQ (CAR L1) L2)
            (UNIØN (CDR L1) L2))
        (T (CØNS (CAR L1)
                 (UNIØN (CDR L1) L2))) ))

(DE INTER (L1 L2)
    (CØND
        ((NULL L1) NIL)
        ((MEMQ (CAR L1) L2)
            (CØNS (CAR L1)
                  (INTER (CDR L1) L2)))
        (T (INTER (CDR L1) L2)) ))
```

Voici, en prime la fonction (PRØCAR L1 L2) et son assistante PRØCA2 qui livre le PRØDUIT CARTESIEN des listes L1 et L2.

```
(DE PRØCAR (L1 L2)
    (CØND
        ((NULL L1) NIL)
        (T (NCØNC (PRØCA2 (CAR L1) L2)
                  (PRØCAR (CDR L1) L2))) ))

(DE PRØCA2 (L1 L2)
    (CØND
        ((NULL L2) NIL)
        (T (CØNS (LIST L1 (CAR L2))
                 (PRØCA2 L1 (CDR L2))) )))
```

Revoir NCØNC page 75.

exemple : (PRØCAR '(A B C) '(1 2 3)) livre :

→ ((A 1)(A 2)(A 3)(B 1)(B 2)(B 3)(C 1)(C 2)(C 3))

on constatera avec plaisir que :

(PRØCAR NIL '(A B)) → NIL

et (PRØCAR '(A B) NIL) → NIL

- CØNCLUSiØN -

Le lecteur aura à présent une certaine idée de LISP en particulier et de la programmation en général. Cette idée me paraît pour l'instant suffisante pour une initiation. Il reste toutefois beaucoup de choses à voir en LISP. Ce sera pour une autre fois. Je crois utile maintenant d'aligner dans ce qui suit un certain nombre de programmes de difficultés, d'intérêts, et d'actualités croissantes. J'en profiterai pour exposer en passage quelques nouvelles notions (et fonctions).

Ces programmes, passés à l'épreuve d'un VRAI ordinateur, je recommande au lecteur de les étudier soigneusement, de les modifier, de s'en servir enfin dans ses PRØPRES programmes.

Pour mieux connaître le langage LISP, je conseille de lire les références que voici :

John Mc CARTHY : "Recursive Functions of Symbolic Expressions and their Computation by Machine" CACM Avril 1963

Mc CARTHY et al. : "LISP 1.5 Programmer's Manual" MIT Press. 1965

Clark WEISSMAN : "LISP 1.5 Primer". DICKENSØN 1967

W.D. MAURER : "The Programmer's Introduction to LISP" Macdonald. 1972

J.L. WHITE : "An Interim LISP User's Guide" MIT. Artificial Intelligence Memo nº 190. Mars 1970.

Joel MØSES : "The Function of FUNCTIØN in LISP" MIT. AI-199 Juin 1970

D.G. BØBRØW : "Requirement for Advanced Programming Systems for List Processing" CACM Juillet 1972.

La fonction DIFFL construit la liste des différences des éléments, puis 2 à 2 de ses arguments L1 et L2. Je suppose que L1 et L2 seront des listes de NOMBRES, de même longueur.

Cette liste de différences sera construite progressivement dans RES. Je boucle → (WHILE L1 ...) tant que L1 contient encore des éléments. C'est l'excellente fonction NEXTL (cf. page 53) qui 1°/ ramène les CAR de L1 et L2
2°/ me fait avancer dans L1 et L2.

On aura progressivement dans RES, pour le 1er exemple :

RES	L1	L2
()	(3 5 7 4)	(1 4 5 1)
(2)	(5 7 4)	(4 5 1)
(2 1)	(7 4)	(5 1)
(2 1 2)	(4)	(1)
(2 1 2 3)	()	()

Les différences des CAR de L1 et L2 :

3 - 1
5 - 4
7 - 5
4 - 1

sont placés à la fin de la liste RES grâce à la fonction NCONC1 (cf. page 75).

Lorsque je sors de la boucle WHILE, RES est constitué. Je le ramène alors comme VALEUR de la fonction DIFFL.

Une liste de SOMMES, de PRODUITS, ou de tout autre opération arithmétique sera constituée de la même façon.

```
(DE DIFFL (L1 L2)
   (PROG (RES)
      (SETQ RES NIL)
      (WHILE L1
         (SETQ RES (NCONC1 RES
            (DIFFER (NEXTL L1) (NEXTL L2)))))
      (RETURN RES)))
 DIFFL
```

```
(DIFFL '(3 5 7 4) '(1 4 5 1))
 (2 1 2 3)

(DIFFL '(1 2 3 4) '(4 3 2 1))
 (-3 -1 1 3)
```

Ceci a été imprimé par la machine, mais n'a pas été tapé sur les cartes. C'est la réponse de LISP à

(DE DIFFL ...)

LISP indique ainsi qu'il a compris que la fonction DIFFL était définie.

Voici 2 façons différentes d'inverser une liste. La première est récursive : il y a un appel de INVERSER dans le corps de la fonction INVERSER ; la seconde ne l'est pas, et utilise un WHILE.

On notera une particularité de la première fonction INVERSER : elle comporte 2 arguments : (DE INVERSER (L1 L2) ...)
ØR, j'appelle INVERSER dans les exemples, avec 1 seul argument ce : (INVERSER '(A B C D)).

Voici pourquoi. C'est un principe, dans les LISPs contemporains d'initialiser automatiquement à NIL les arguments manquants à l'appel. On aura ainsi :

L1	L2
(A B C D)	()

L'idée c'est, progressivement de placer en tête de L2, les éléments trouvés en avançant dans L1. Puis quand ya plus d'éléments dans L1 (cf : le ((NULL L1) L2)), on rentre L2.
Ça se passe à peu près ainsi :

L1	L2
(A B C D)	()
(B C D)	(A)
(C D)	(B A)
(D)	(C B A)
()	(D C B A)

La seconde fonction INVERSER n'offre aucune difficulté.
Dans les 2 cas, on inverse bien les éléments de L1, mais si un de ces éléments est une liste, cette liste n'est pas inversée.
cf : (le rape (est mort)) → ((est mort) rape le)

Dans le 3ème exemple, L1 est identiquement lue de gauche → droite et de droite → gauche. On dit que c'est un PALINDRØME. Il y a quelques jolis palindromes dans le "Guide des jeux d'esprits" de Jacques Bens, chez Albin Michel pages 113 - 116.

L AME DES UNS IAMMS N USE DE MAL

(Anonyme, Moyen Age)

```
(DE INVERSER (L1 L2)
   (COND
      ((NULL L1) L2)
      (T (INVERSER (CDR L1)
                   (CONS (CAR L1) L2))) ))
 INVERS
```

```
(DE INVERSER (L)
 (PROG (RES)
  (WHILE L
      (SETQ RES (CONS (NEXTL L) RES)))
   (RETURN RES)))
 INVERS
```

```
(INVERSER '(A B C D))
 (D C B A)
```

```
(INVERSER '(LE PAPE (EST MORT)))
 ((EST MORT) PAPE LE)
```

```
(INVERSER '(E M I L E  N U  A  U N E  L I M E))
 (E M I L E N U A U N E L I M E)
```

La fonction MDYCK doit reconnaître si son argument L est, ou n'est pas un <u>mot de Dyck</u>.

L'idée de "mot de DYCK" c'est la généralisation de l'idée intuitive d'une expression CORRECTEMENT PARENTHESÉE.

exemple :		
	()	ØK
	)(	NØN
	((()())	ØK
	()()()	ØK
	(())	NØN

C'est fort important. On trouve ça partout dans la nature. Ça sous-tend la théorie linguistique des langages CONTEXT-FREE. Un programme LISP, ou une fonction LISP bien parenthésé est un mot de DYCK (restreint toutefois).

L'idée c'est d'avancer dans le mot, après avoir mis un compteur K à 0. Si je trouve une ouvrante, j'ajoute 1 au compteur, si je trouve une fermante, j'ôte 1. A tout moment, si K devient négatif, c'est que le mot est faux (ya trop de fermantes). Quand j'arrive à la fin du mot, si $K \neq 0$ le mot est faux (ya trop d'ouvrantes), si $K = 0$, le mot est correct.

Ici les éléments du mot testé sont 1 et −1.

1 représente la "("
−1 représente la ")"

ex : pour le 1er exemple, je décode ainsi :

((()()) () correct.

Pour en savoir plus long, on peut lire la page 17 de "Notions sur les grammaires formelles" de Maurice GROSS et Andre LENTIN chez Gauthier-Villars, ou la (difficile) :
"Theorie algébriques des langages context-free" par N. CHOMSKY et Marcel P. SCHUTZENBERGER dans le n° de Mars 1963 de la revue LANGAGES (n° 3).

```
(DE MDYCK (L)
        (PROG (K)
              (SETQ K 0)
      RE      (COND
                ((NULL L)
                  (RETURN
                    (COND ((ZEROP K) 'OK)
                          (T 'NON))))
                ((EQ (CAR L) 1)
                  (SETQ K (ADD1 K)))
                ((EQ (CAR L) -1)
                  (SETQ K (SUB1 K))
                  (COND
                    ((LT K 0) (RETURN 'NON)))))
              (SETQ L (CDR L))
              (GO RE) ))
 MDYCK


(MDYCK '(1 1 -1 1 -1 -1 1 -1))
 OK


(MDYCK '(1 1 1 -1))
 NON


(MDYCK '(1 1 -1 -1 -1))
 NON


(MDYCK '(1 1 1 1 -1 -1 -1 1 -1 -1))
 OK
```

Voici 2 suites ordonnées (ici par ordre croissant) de nombres, L1 et L2.

exemple : L1 = 1 14 21 22 72 87

L2 = 0 6 8 16 21 35 37 45

Je me propose de combiner ces 2 suites en une seule, RES, qui sera ordonnée de la même façon.

ici : RES = 0 1 6 8 14 16 21 21 22 35 37 45 72 87

Je dirai que j'ai opéré la FUSION, ou encore l'INTERCLASSEMENT de L1 et L2 (noter que L1 et L2 peuvent avoir des longueurs différentes).

Cette opération est fort utile pour mettre à jour des fichiers (voir page 79). exemple : un fichier de patineurs ordonné par âges croissants.

L'idée c'est : j'ai donc L1 et L2. Je construis pas à pas une 3ème suite, ici RES en ajoutant à la fin (voir NCONC1 page 75) de la partie déjà construite de RES, un élément de L1 ou de L2 ainsi choisi :

. je compare les 1ers éléments (CAR) de L1 et L2. J'extrais le plus petit des deux, avançant ainsi dans L1 ou L2 (NEXTL), et je le place à la fin de RES.

A tout moment si l'une des 2 suites, L1 ou L2 devient vide, je concatène (voir page 75) à RES : L1 si L2 est vide, L2 si L1 est vide, et je ramène RES comme valeur de la fonction INTERCLASSER.

Dans le 1er exemple, on aura ainsi :

L1	L2	RES
(1 4 7)	(0 3 5)	()
(1 4 7)	(3 5)	(0)
(4 7)	(3 5)	(0 1)
(4 7)	(5)	(0 1 3)
(7)	(5)	(0 1 3 4)
(7)	()	(0 1 3 4 5)
		→ (0 1 3 4 5 7)

```
(DE INTERCLASSER (L1 L2)
   (PROG (RES)
              (SETQ RES NIL)
      HOP     (COND
                ((NULL L1) (RETURN (NCONC RES L2)))
                ((NULL L2) (RETURN (NCONC RES L1))) )
              (SETQ RES (NCONC1 RES
                  (COND
                    ((LT (CAR L1) (CAR L2)) (NEXTL L1))
                    (T (NEXTL L2)) )))
              (GO HOP) ))
 INTERC
```

```
(INTERCLASSER '(1 4 7) '(0 3 5))
 (0 1 3 4 5 7)
```

```
(INTERCLASSER '(-2 5 21 37) '(2 3 4 18))
 (-2 2 3 4 5 18 21 37)
```

```
(INTERCLASSER '(1515 1789 1944) '(1870 1944 1968 2001))
 (1515 1789 1870 1944 1944 1968 2001)
```

La fonction BINDEC illustre l'algorithme classique de CONVERSION d'un nombre binaire en un nombre décimal. Triple abus de langage.

1/ Je n'opère pas une conversion entre nombres, mais entre des REPRESENTATIONS du même nombre, entité idéale.

exemple : "5273" est la représentation dite DÉCIMALE de :

$$\underline{5} * 10^3 + \underline{2} * 10^2 + \underline{7} * 10^1 + \underline{3} * 10^0 .$$

Dans ce cas, il est courant d'appeler 10 la BASE de la représentation. De même, "5273" est une représentation de :

$$((\underline{5} * 10 + \underline{2}) * 10 + \underline{7}) * 10 + \underline{3}$$

Ce dernier calcul a l'avantage de pouvoir être effectué en lisant les chiffres du nombre de GAUCHE ⟶ DROITE, sans avoir à connaître à l'avance combien de chiffres vont suivre. On nomme ceci "représentation de HÖRNER". En pratique : j'initialise un compteur à 0. Puis, tant qu'il se présente un chiffre, je multiplie le compteur par 10 et j'ajoute le chiffre. En général :

soit b une base, a_i un chiffre $\in [0, b-1]$, on aura ceci :

$a_n a_{n-1} \ldots a_2 a_1 a_0$ donnera, en représentation de Hörner :

$(((\ldots (a_n * b + a_{n-1}) * b + \ldots) * b + a_2) * b + a_1) * b + a_0$

2/ Je n'ai donc pas affaire à un "nombre binaire" mais à la représentation d'un nombre en base 2. Ici $b = 2$, et les chiffres $\in [0, 1]$. Pour la fonction BINDEC je représente la suite des chiffres d'un "nombre binaire" par une liste de "chiffres binaires", 0 ou 1. exemple : $(1\ 0\ 1) \rightarrow (\underline{1} * 2 + \underline{0}) * 2 + \underline{1}$

3/ même chose pour les "nombres décimaux"

On aura par exemple, pour le 4ème appel :

L	RES
(1 1 0 1 1)	0
(1 0 1 1)	1 = 2*0 + 1
(0 1 1)	3 = 2*1 + 1
(1 1)	6 = 2*3 + 0
(1)	13 = 2*6 + 1
( )	27 = 2*13 + 1

```
(DE BINDEC (L)
   (PROG (RES)
      (SETQ RES 0)
      (WHILE L
         (SETQ RES (PLUS (TIMES RES 2)
                         (NEXTL L))))
      (RETURN RES)))
 BINDEC



(BINDEC '(1 0))
 2


(BINDEC '(1 0 1))
 5


(BINDEC '(1 1 1 1))
 15


(BINDEC '(1 1 0 1 1))
 27


(BINDEC '(1 1 1 1 1 0 0 0 0 1))
 993
```

DECBIN illustre l'algorithme classique de conversion d'un nombre entier décimal en un nombre binaire (toujours par abus de langage). J'utilise les mêmes représentations que pour la fonction BINDEC (progs 101-102).

exemple : $29_{10} = 11101_2$. Les petits $_{10}$ et $_2$ indiquent que les nombres sont représentés en base 10 et en base 2. L'idée c'est :

```
29 | 2
 1 | 14 | 2
   0 |  7 | 2
     1 |  3 | 2
       1 |  1 | 2
         1 |  0
```

Je m'arrête lorsque le quotient devient 0. La suite INVERSÉE des restes de ces divisions successives me donne la représentation binaire de mon nombre.

Point nécessaire ici d'effectuer une telle inversion, puisque CONS accumule naturellement le reste en tête de la liste RES.

exemple avec N = 29

N	RES
29	()
14	(1)
7	(0 1)
3	(1 0 1)
1	(1 1 0 1)
0	(1 1 1 0 1)

Au sujet des conversions, on lira avec profit les pages 280 à 290 de :
"The Art of Computer Programming, Vol 2 : Seminumerical Algorithms" de Donald KNUTH, chez ADDISON-WESLEY.

```
(DE DECBIN (N)
   (PROG (RES)
      (SETQ RES NIL)
      (WHILE (GT N 0)
         (SETQ RES (CONS (REM N 2) RES))
         (SETQ N (QUO N 2)))
      (RETURN RES)))
 DECBIN
```

```
(DECBIN 2)
 (1 0)
```

```
(DECBIN 5)
 (1 0 1)
```

```
(DECBIN 8)
 (1 0 0 0)
```

```
(DECBIN 15)
 (1 1 1 1)
```

```
(DECBIN 27)
 (1 1 0 1 1)
```

```
(DECBIN 993)
 (1 1 1 1 1 0 0 0 0 1)
```

La fonction NBMIRØIR livre en résultat un nombre RES, dont la suite des chiffres est obtenue par inversion de la suite des chiffres de N donné en argument.

ex : 1984 → 4891.

L'idée est, à chaque étape d'extraire le DERNIER chiffre de N et de le placer en tête de RES.

(REM N 10) : permet de connaître le dernier chiffre de N.

(QUØ N 10) : enlève le dernier chiffre de N.

Ainsi, pour le 1er exemple, on obtiendra successivement :

N	RES
572	0
57	2 = 0*10 + 2
5	27 = 2*10 + 7
0	275 = 27*10 + 5

On notera que NBMIRØIR est une application directe des idées illustrées par les fonctions BINDEC et DECBIN (pages 101 à 104).

Une façon particulièrement simple de reconnaître si la suite des chiffres d'un nombre constitue ou non un palindrome serait

```
(DE ISMIRØIR (N)
    (EQ N (NBMIRØIR N )))
```

```
(DE NBMIROIR (N)
   (PROG (RES)
      (SETQ RES 0)
      (WHILE (GT N 0)
         (SETQ RES (PLUS (TIMES RES 10)
                         (REM N 10)))
         (SETQ N (QUO N 10)))
      (RETURN RES)))
 NBMIRO



(NBMIROIR 572)
 275



(NBMIROIR 12345)
 54321



(NBMIROIR 65432)
 23456



(NBMIROIR 55555)
 55555
```

La fonction RACZE calcule la racine carrée "entière" du nombre N.
Sa construction est fondée sur une relation liant les carrés d'entiers, et les sommes d'entiers impairs consécutifs.

$1^2 = 1 = 1$
$2^2 = 4 = 1 + 3$
$3^2 = 9 = 1 + 3 + 5$
$4^2 = 16 = 1 + 3 + 5 + 7$
$5^2 = 25 = 1 + 3 + 5 + 7 + 9$

En général : $n^2 = 1 + 3 + \ldots + (2n-1)$

Ainsi, calculer le carré d'un entier n sans utiliser la multiplication est facile : 1° K←0, P←1 (je note ici "K←0" l'écriture LISP correcte "(SETQ K 0)")

2° si N=0 STOP : le carré est dans K.

3° K←K+P, P←P+2, N←N−1, vers 2°.

Ainsi, pour N=5, on obtient successivement :

N	K	P
5	0	1
4	1	3
3	4	5
2	9	7
1	16	9
0	[25]	11

INVERSEMENT : pour trouver $\sqrt{N}$, il me suffit de compter combien de nombres impairs successifs je puis ajouter sans dépasser N.

Donc, avec N donné, trouver $\sqrt{N}$

1° P←1, K←0
2° N←N−P
3° si N<0 STOP : $\sqrt{N}$ est dans K.
4° P←P+2, K←K+1, vers 2°.

Ainsi, avec par exemple N = 27, on aura successivement :

N	K	P
27	0	1
26	1	3
23	2	5
18	3	7
11	4	9
2	[5]	11
−9		STOP

Dans RACZE, j'utilise la fonction (NOT obj), qui est parfaitement équivalente à NULL, i.e.

(NOT obj) → { T si valeur de obj = NIL ; NIL si valeur de obj ≠ NIL }

```
(DE RAC2E (N)
   (PROG (P K)
      (SETQ P 1)
      (SETQ K 0)
      (WHILE (NOT (LT (SETQ N (DIFFER N P)) 0))
         (SETQ P (PLUS P 2))
         (SETQ K (ADD1 K)) )
      (RETURN K) ))
 RAC2E

(RAC2E 9)
 3

(RAC2E 8100)
 90

(RAC2E 15)
 3

(RAC2E 271)
 16
```

La fonction récursive DELREP construit la liste des atomes distincts de sa liste-argument L.

L'idée c'est : tant que la liste n'est pas vide, extraire son 1er élément. Si cet élément n'a pas d'occurrence dans la suite de L, coller cet élément en tête de la liste-résultat en construction. Si cet élément a au moins une occurrence dans la suite de L, l'ignorer. Puis avancer dans L.

Nous avons vu pages 64 et 50 comment construire la fonction (MEMQ at l) qui donne $\begin{cases} T \text{ si at} \in l \\ NIL \text{ si at} \notin l \end{cases}$.

Nous n'avons pas à définir cette fonction. LISP la met automatiquement à notre disposition, au même titre que CAR, CDR, EQ, etc.. On dit que MEMQ est une fonction STANDARD.

Ainsi, avec, par exemple :

(DELREP '(A B A B C)) on aura successivement :

↓ (DELREP '(A B A B C))

↓ (DELREP '(B A B C)) puisque A ∈ (B A B C), c'est ce que teste précisément MEMQ.

↓ (DELREP '(A B C)) puisque B ∈ (A B C).

↓ (CONS 'A (DELREP '(B C))) puisque A ∉ (B C).

↓ (CONS 'A (CONS 'B (DELREP '(C)))) puisque B ∉ (C).

↓ (CONS 'A (CONS 'B (CONS 'C (DELREP NIL))))

puisque C ∉ () et pour cause !

↓ (CONS 'A (CONS 'B (CONS 'C NIL)))

↓ (CONS 'A (CONS 'B '(C)))

↓ (CONS 'A '(B C))

↓ (A B C).

```
(DE DELREP (L)
  (COND
       ((NULL L) NIL)
       ((MEMQ (CAR L) (CDR L))
          (DELREP (CDR L)))
       (T (CONS (CAR L)
                (DELREP (CDR L))))))
 DELREP



(DELREP '(JE JE JE JE JE JE JE NE NE NE NE NE NE
      BEGAYE PAS PAS PAS PAS PAS))
 (JE NE BEGAYE PAS)
```

La fonction récursive PØLØNISER accepte comme argument une expression de structure : [atome ou (expression$_1$ opérateur expression$_2$)]

et livre en résultat une autre expression de structure [atome ou (opérateur expression$_1$ expression$_2$)]

Notez que les "opérateurs", ici + , − , * , / sont des atomes comme les autres.

PØLØNISER illustre bien les raisons pour lesquelles LISP est amené à offrir des fonctions récursives ; à savoir dans tous les cas où on doit traiter une structure S, comportant des sous-structures $S_1, S_2, \ldots S_n$ de structure analogue à S : on re-appelle la fonction sur les sous-structures ainsi dégagées.

PØLØNISER avale donc une expression parenthésée et recrache une expression polonaise préfixe.

On construit aisément une fonction DEPØL qui avale une expression préfixe et recrache l'expression parenthésée, exactement sur le même principe :

```
(DE  DEPØL  (E)
   (CØND
      ((ATØM  E) E)
      (T (LIST  (DEPØL (CADR E))
                (CAR E)
                (DEPØL (CADDR E)) )) ))
```

exemple : (DEPØL '(* A (+ B C))) ⇝ (A * (B + C))

```
(DE POLONISER (E)
  (COND
       ((ATOM E) E)
       (T (LIST (CADR E)
                (POLONISER (CAR E))
                (POLONISER (CADDR E)) )) ))
 POLONI
```

```
(POLONISER 'A)
 A
```

```
(POLONISER '(A + B))
 (+ A B)
```

```
(POLONISER '(A * (B + C)))
 (* A (+ B C))
```

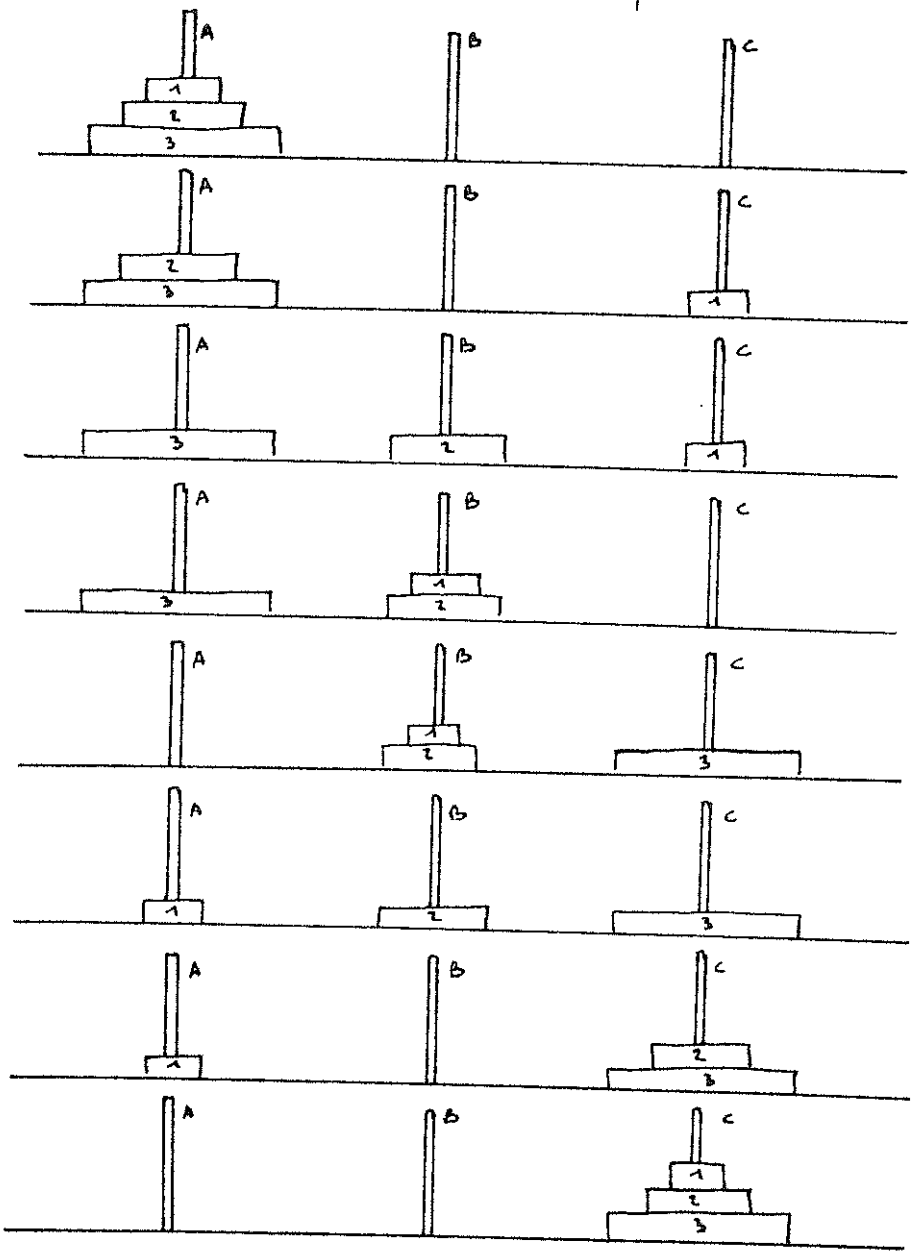
```
(POLONISER '((A - (B + C)) * (D / E)))
 (* (- A (+ B C)) (/ D E))
```

Lors de la Création du monde, Dieu fit apparaître sous l'Himalaya, 64 disques d'or de diamètres décroissants, empilés sur une aiguille de diamant. Il plaça également à proximité 2 autres aiguilles de diamant, toutes deux vides. A cette occasion, Il créa également des prêtres et leur donna la tâche de déplacer les disques de leur aiguille originelle vers une seconde aiguille, en utilisant si nécessaire la troisième, comme lieu de stockage temporaire. Lorsque les 64 disques se retrouveront empilés par taille décroissante sur la seconde aiguille, le moment de la fin du monde sera arrivé. Les déplacements des disques seront réglés par les règles que voici :

1/ on ne déplace qu'1 disque à la fois.

2/ on ne doit jamais poser un disque sur un disque de diamètre inférieur.

exemple : avec : nombre de disques : N = 3.

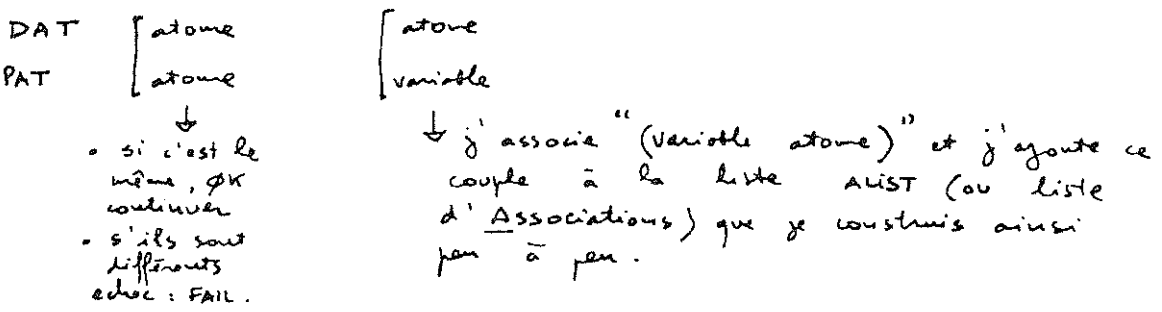


Certains appellent les 3 aiguilles, les TOURS DE HANOÏ.

```
(DE HANOI (N DEPART ARRIVEE INTERMEDIAIRE)
  (COND
       ((GT N 0)
        (HANOI (SUB1 N) DEPART INTERMEDIAIRE ARRIVEE)
        (PRINT (LIST 'DISQUE N 'DE DEPART 'VERS ARRIVEE))
        (HANOI (SUB1 N) INTERMEDIAIRE ARRIVEE DEPART)) ))
 HANOI



(HANOI 1 'A 'C 'B)
 (DISQUE 1 DE A VERS C)
 NIL


(HANOI 2 'A 'C 'B)
 (DISQUE 1 DE A VERS B)
 (DISQUE 2 DE A VERS C)
 (DISQUE 1 DE B VERS C)
 NIL


(HANOI 3 'A 'C 'B)
 (DISQUE 1 DE A VERS C)
 (DISQUE 2 DE A VERS B)
 (DISQUE 1 DE C VERS B)
 (DISQUE 3 DE A VERS C)
 (DISQUE 1 DE B VERS A)
 (DISQUE 2 DE B VERS C)
 (DISQUE 1 DE A VERS C)
 NIL


(HANOI 4 'A 'C 'B)
 (DISQUE 1 DE A VERS B)
 (DISQUE 2 DE A VERS C)
 (DISQUE 1 DE B VERS C)
 (DISQUE 3 DE A VERS B)
 (DISQUE 1 DE C VERS A)
 (DISQUE 2 DE C VERS B)
 (DISQUE 1 DE A VERS B)
 (DISQUE 4 DE A VERS C)
 (DISQUE 1 DE B VERS C)
 (DISQUE 2 DE B VERS A)
 (DISQUE 1 DE C VERS A)
 (DISQUE 3 DE B VERS C)
 (DISQUE 1 DE A VERS B)
 (DISQUE 2 DE A VERS C)
 (DISQUE 1 DE B VERS C)
 NIL
```

La fonction MATCH illustre un concept qui a pris ces dernières années en programmation une importance extrême : le PATTERN-MATCHING. (abbrévié P-M)
Le pattern-matching est une généralisation de l'idée de tester si 2 listes de même longueur comportent les mêmes atomes à la même place.

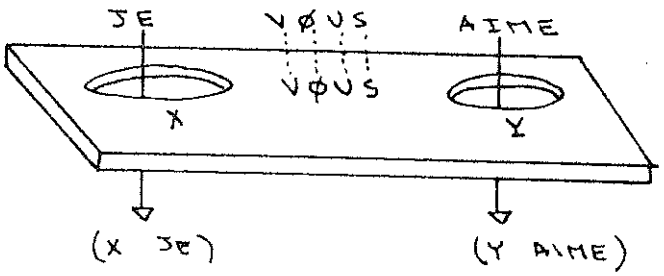
On distinguera donc : un ITEM : liste d'atomes ordinaires.
un PATTERN : une liste où certains atomes ont été distingués comme étant des "variables de pattern". Ici, je reconnais si un objet est une variable de pattern grâce à la fonction ISVAR. Le pattern a une action très semblable à celle d'un FILTRE, avec des "trous" qui sont ses variables.
Dans la fonction MATCH, le pattern est PAT, l'item est DAT. S'ils sont de longueur différente, je considère le P-M comme raté et je ramène "FAIL" en valeur. S'ils sont de même longueur, je compare les 1ers éléments de PAT et de DAT.



S'il n'y a pas échec, je continue en avançant à la fois dans PAT et DAT.
Si j'arrive sans succomber à la fin de PAT (et donc de DAT) je ramène ALIST comme valeur du pattern-matching.

Ainsi dans le 3e exemple on aura :

PAT	DAT	P	D	ALIST
(X VOUS Y)	(JE VOUS AIME)			()
(VOUS Y)	(VOUS AIME)	X	JE	((X JE))
(Y)	(AIME)	VOUS	VOUS	
()	()	Y	AIME	((Y AIME)(X JE))



Le pattern-matching est un des mécanismes de base de langages de programmation comme SNOBOL, PLANNER, QA4 et CONNIVER.
On constatera que LENGTH (voir pages 40, 47, 61, 85) est une fonction LISP standard (je n'ai pas à la définir).

```
(DE MATCH (PAT DAT)
   (PROG (P D ALIST)
      (SETQ ALIST NIL)
      (COND
        ((NEQ (LENGTH PAT) (LENGTH DAT))
          (RETURN 'FAIL))
        (T (WHILE PAT
             (SETQ P (NEXTL PAT))
             (SETQ D (NEXTL DAT))
             (COND
                ((ISVAR P)
                  (SETQ ALIST (CONS (LIST P D) ALIST)))
               ((NEQ P D) (RETURN 'FAIL))))
           (RETURN ALIST) ))))
 MATCH

(DE ISVAR (OB)
   (MEMQ OB '(V W X Y Z)))
 ISVAR


(MATCH '(JE VOUS X) '(JE VOUS AIME))
 ((X AIME))


(MATCH '(JE VOUS AIME) '(JE NE VOUS AIME PAS))
 FAIL


(MATCH '(X VOUS Y) '(JE VOUS AIME))
 ((Y AIME) (X JE))


(MATCH '(X EST Y PAR Z) '(PAUL EST TUE PAR JEAN))
 ((Z JEAN) (Y TUE) (X PAUL))


(MATCH '(W X Y Z) '(DIEU EST PLUS SAVANT))
 ((Z SAVANT) (Y PLUS) (X EST) (W DIEU))


(MATCH '(W X PAS Z) '(DIEU EST PLUS SAVANT))
 FAIL
```

La fonction INSTANCE admet 2 arguments : un pattern (voir p. 115) et une A-liste ou dictionnaire, liste de couples de la forme :

(variable valeur)

INSTANCE construit une liste RES identique au pattern, à ceci près que lorsque je trouve dans le pattern un élément p tel qu'il existe dans le dictionnaire un couple (p objet), je substitue objet dans RES. Si l'élément p ne se trouve pas dans le dictionnaire, je prends p lui-même.

Je construit RES en ajoutant progressivement les éléments à la queue, par NCONC1. La recherche dans le dictionnaire se fait grâce à la fonction ASSOC (voir pages 67 et 30)

exemple : (ASSOC 'X '((Y MOI) (X JE) (Z TU))) ⇝ (X JE)

(ASSOC 'JOUR '((X DUR) (Y MOU))) ⇝ NIL

On peut obtenir des effets intéressants en combinant ainsi MATCH et INSTANCE.

exemples :

```
(INSTANCE '(POUR QUOI ETES VOUS X ?)
          (MATCH '(JE SUIS X)
                 '(JE SUIS TRISTE)))
⇝ (POUR QUOI ETES VOUS TRISTE ?)

(INSTANCE '(Y EST TUE PAR X)
          (MATCH '(X TUE Y)
                 '(PAUL TUE JEAN)))
⇝ (JEAN EST TUE PAR PAUL)
```

C'est sur une telle combinaison (plus complexe) qu'est fondé le célèbre programme ELIZA, de Joseph WEIZENBAUM, capable de tenir une conversation plausible (on s'y trompe, en oubliant après un certain temps, qu'on cause avec un programme) dans une langue naturelle (anglais ou allemand ou gallois).

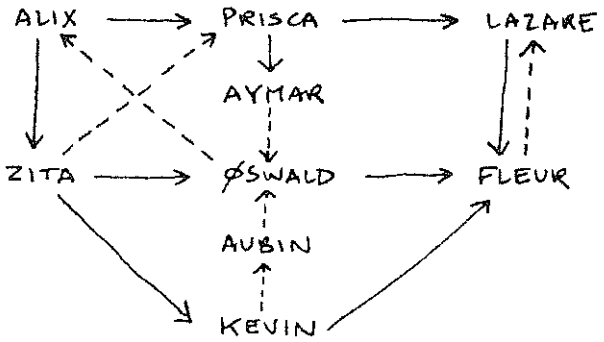
"A computer program for the study of natural language communication between man and machine" J. WEIZENBAUM

CACM 1966, 9, 36-45.

```
(DE INSTANCE (PAT ALISTE)
   (PROG (RES P X)
      (SETQ RES NIL)
      (WHILE PAT
             (SETQ P (NEXTL PAT))
             (SETQ X (ASSOC P ALISTE))
             (SETQ RES (NCONC1 RES
                 (COND
                   (X (CADR X))
                   (T P) ))) )
             (RETURN RES) ))
 INSTAN

(DE ASSOC (OB L)
   (COND
      ((NULL L) NIL)
      ((EQ (CAAR L) OB) (CAR L))
      (T (ASSOC OB (CDR L)))))
 ASSOC



(INSTANCE '(X VOUS Y) '((X JE) (Y AIME)))
 (JE VOUS AIME)


(INSTANCE '(X EST Y PAR Z) '((X PAUL) (Y TUE) (Z JEAN)))
 (PAUL EST TUE PAR JEAN)


(INSTANCE '(X VOUS Y) '((X (JE NE)) (Y (AIME PAS))))
 ((JE NE) VOUS (AIME PAS))


(INSTANCE '(CIEL X VOUS ETES ENCORE Y) '((X IGOR) (Y SAOUL)))
 (CIEL IGOR VOUS ETES ENCORE SAOUL)
```

Imaginons que nous avons une tragédie, un sombre drame elizabethain, mettant en jeu entre de nombreux personnages des relations fatales d'amour et de haine, dont voici le sombre réseau :

→ aime
--→ hait



BRICE ⇄ NADEGE

Ce réseau, je le représente par une collection de triplets (listes à 3 éléments) de la forme :

(nom {AIME / HAIT} nom)

et, cette collection, je l'associe avec la variable CONTEXT. On notera que cette variable n'est pas déclarée, ni dans un PROG, ni en argument de fonction; on dit dans ce cas qu'il s'agit d'une "variable GLOBALE".

A ce contexte, j'aimerai poser des questions telles que

1. qui aime LAZARE ?
2. ØSWALD hait qui ?
3. qui aime qui ?

etc.

et recevoir en réponse :

1. PRISCA aime LAZARE
2. ØSWALD hait ALIX
3. NADEGE aime BRICE, BRICE aime NADEGE, KEVIN aime FLEUR, ZITA aime KEVIN, etc.

C'est précisément l'objet de la fonction FETCH.

La question se présente sous la forme d'un PATTERN, et j'utilise la fonction MATCH, que nous avons définie page 116.

```
(DE FETCH (PAT)
   (PROG (RES C)
      (SETQ X CONTEXT)
      (WHILE X
         (SETQ C (NEXTL X))
         (COND
           ((NEQ (MATCH PAT C) 'FAIL)
             (SETQ RES (CONS C RES)))))
      (RETURN RES)))
 FETCH

(SETQ CONTEXT '(
               (ALIX AIME PRISCA)
               (ALIX AIME ZITA)
               (PRISCA AIME LAZARE)
               (PRISCA AIME AYMAR)
               (LAZARE AIME FLEUR)
               (FLEUR AIME ALIX)
               (FLEUR HAIT LAZARE)
               (ZITA HAIT PRISCA)
               (ZITA AIME OSWALD)
               (ZITA AIME KEVIN)
               (AYMAR HAIT OSWALD)
               (OSWALD AIME FLEUR)
               (OSWALD HAIT ALIX)
               (KEVIN AIME FLEUR)
               (KEVIN HAIT AUBIN)
               (AUBIN HAIT OSWALD)
               (BRICE AIME NADEGE)
               (NADEGE AIME BRICE)
      ))
 ((ALIX AIME PRISCA) (ALIX AIME ZITA) (PRISCA AIME LAZARE) (
PRISCA AIME AYMAR) (LAZARE AIME FLEUR) (FLEUR AIME ALIX) (FLEUR
HAIT LAZARE) (ZITA HAIT PRISCA) (ZITA AIME OSWALD) (ZITA AIME
KEVIN) (AYMAR HAIT OSWALD) (OSWALD AIME FLEUR) (OSWALD HAIT
ALIX) (KEVIN AIME FLEUR) (KEVIN HAIT AUBIN) (AUBIN HAIT OSWALD
) (BRICE AIME NADEGE) (NADEGE AIME BRICE))
```

le processus de construction de la réponse est simple : je "matche" le pattern avec tous les triplets du contexte, si le matching réussit, j'ajoute le triplet à la réponse, et je continue dans tous les cas.

La réponse est donc construite à coup de CONS dans la variable RES. On notera que celle ci n'est pas initialisée. Ceci tient à ce que LISP initialise automatiquement les variables de la liste de variables locales d'un PROG à NIL. Donc, on entrant dans le PROG on a automatiquement :

RES	C
NIL	NIL

J'aurai pu le dire plus tôt. Mais je tenais beaucoup à souligner l'importance de la notion d'initialisation.

On appelle également BASE DE DONNÉE (data base) ce grand ensemble d'items qu'on interroge à coup de pattern-matching. Quand on a une base de données interrogeable de façon semblable à FETCH : "ramener tous les items tels que ...", on dit qu'on a affaire à une MÉMOIRE ASSOCIATIVE.

Retenez qu'une mémoire c'est
1°/ un ensemble de données stockées
2°/ un mode spécifique d'interrogation de ces données.

Plusieurs langages de programmation sont construits autour de ces notions, dont l'application est immédiate, par exemple aux relations de parenté, LEAP, SAIL. On aura alors des triplets de la forme :

(attribut objet valeur)

ex. : (PERE PAUL JEAN)

question : ramener tous les noms des personnes dont JEAN est le père. etc.

Dans l'exemple 8, on notera la curieuse question "(X Y Z)", qui peut s'interpréter comme l'énoncé apocalyptique :

"qu'est-ce qui est le quoi de quoi" QQQQ

ou bien "qui quoi qui" QQQ

```
(FETCH '(ZITA AIME Y))
 ((ZITA AIME KEVIN) (ZITA AIME OSWALD))


(FETCH '(X Y FLEUR))
 ((KEVIN AIME FLEUR) (LAZARE AIME FLEUR))


(FETCH '(X HAIT Y))
 ((AUBIN HAIT OSWALD) (KEVIN HAIT AUBIN) (OSWALD HAIT ALIX) (
AYMAR HAIT OSWALD) (ZITA HAIT PRISCA) (FLEUR HAIT LAZARE))


(FETCH '(OSWALD Y Z))
 ((OSWALD HAIT ALIX))


(FETCH '(X Y OSWALD))
 ((AUBIN HAIT OSWALD) (AYMAR HAIT OSWALD) (ZITA AIME OSWALD))


(FETCH '(FLEUR X LAZARE))
 ((FLEUR HAIT LAZARE))


(FETCH '(X AIME Z))
 ((NADEGE AIME BRICE) (BRICE AIME NADEGE) (KEVIN AIME FLEUR) (
ZITA AIME KEVIN) (ZITA AIME OSWALD) (FLEUR AIME ALIX) (LAZARE
AIME FLEUR) (PRISCA AIME AYMAR) (PRISCA AIME LAZARE) (ALIX
AIME ZITA) (ALIX AIME PRISCA))


(FETCH '(X Y Z))
 ((NADEGE AIME BRICE) (BRICE AIME NADEGE) (AUBIN HAIT OSWALD) (
KEVIN HAIT AUBIN) (KEVIN AIME FLEUR) (OSWALD HAIT ALIX) (AYMAR
HAIT OSWALD) (ZITA AIME KEVIN) (ZITA AIME OSWALD) (ZITA HAIT
PRISCA) (FLEUR HAIT LAZARE) (FLEUR AIME ALIX) (LAZARE AIME
FLEUR) (PRISCA AIME AYMAR) (PRISCA AIME LAZARE) (ALIX AIME
ZITA) (ALIX AIME PRISCA))


(FETCH '(X AIME BRICE))
 ((NADEGE AIME BRICE))


(FETCH '(Y AIME NADEGE))
 ((BRICE AIME NADEGE))
```

c'est charmant.